

NPS-52PW72071A

United States Naval Postgraduate School

//



FUNCTIONAL PROGRAM MODULES (FPMs)

AND

DIGITAL SYSTEMS DESIGN

by

V. MICHAEL POWERS

20 JULY 1972

Approved for public release; distribution unlimited.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral M. B. Freeman
Superintendent

M. U. Clauser
Provost

ABSTRACT:

Design of digital electronic systems for a range of applications such as the functions of small ships requires a unified approach which does not make a priori choices between hardware and software. Throughout the several levels of digital system structure, the design procedure is portrayed as a task of preparing a program and translating the program into successively different languages. One language is the language of Functional Program Modules. These modules are versatile, effective, program control and operative modules which can easily be configured (programmed) and can be used with methods which automatically produce assembly and maintenance information. Examples, extensions and applications of FPMs are discussed.

This is the final report of a task funded by the Naval Electronics Laboratory Center under Work Request No. 2-9104, amendment number one, 17 April 1972.

NPS-52PW72071A

20 July 1972

FOREWORD

This report was prepared in an effort to describe some of the most important concepts which are fundamental to the design of digital information systems. The report grew from an answer to a question from Mr. Norman Tinklepaugh, and was meant more to provide a background and a base for further discussion than as an immediately applicable product. Time to develop the answers was supported by funds from projects under the cognizance of him and the Division Head of the Naval Electronics Laboratory Center's Advanced Modular Concepts Division (Code 4300), Mr. William Dejka. I thank both these men for their suggestions and assistance.

V. Michael Powers

FUNCTIONAL PROGRAM MODULES (FPMS)

AND DIGITAL SYSTEMS DESIGN

TABLE OF CONTENTS

Foreword	ii
Table of Contents	iii
List of Figures	v
1. Introduction	1
1.1 Scope	1
1.2 Small ships electronics	2
1.3 Soft, firm and hard wares	3
2. Levels of organization	6
2.1 Levels of design	6
2.2 Programs	8
3. Program preparation (design)	10
3.1 Traditional software generation	14
3.2 Life cycle considerations	17
3.3 Means of program preparation	20
4. Functional Program Modules (FPMS)	21
4.1 Design as configuration	21
4.1.1. Standards as design	24
4.1.2. Design (configuration) aids	25
4.2 Register Transfer Modules (RTMs)	27
4.2.1. Origin of RTMs	27
4.2.2. PDP-16	31
4.2.3. Simulations	32
4.2.4. Other examples	34

4.3. Functional Program Modules (FPMs).	36
5. Applications and conclusions	39
5.1 Applications of Functional Program Modules	39
5.2 Conclusions	40
References	43
Distribution List	44
Form DD1473	47

LIST OF FIGURES

<u>NUMBER</u>	<u>CAPTION</u>	<u>IN SECTION</u>	<u>PAGE</u>
1.	A Cartoon Showing a Robust Design	3.	12
2.	An Example RTM Program	4.3.1	30

FUNCTIONAL PROGRAM MODULES (FPMs)

AND DIGITAL SYSTEMS DESIGN

1. Introduction

1.1. Scope

This report collects for review some thoughts about present and near-future concerns in digital systems design, particularly with respect to module selection and modular design methods.

The discussion focuses on a particular range of families of digital modules which have attractive possibilities for future use in system design. Enough consideration is given to the general design problem, however, to place this discussion in a general context.

A particular concern of the report is to emphasize the commonality between so-called "hardware" and "software" design. In spite of the fact that the two types of activities are considered separate in many organizations today, it is believed that separating the design activities for digital systems into hardware and software is detrimental to the objective of obtaining the best mixture of programming and devices for a given system. Furthermore, the knowledge and procedures involved in both activities are so similar and related that attempts to improve the overall system design process benefit from ignoring the superficial differences. It is recognized that the individual computer programmer need not be a circuit designer, nor vice versa, but the system designer needs knowledge of both.

The later sections of this report discuss a particular set of commercial modules, along with some extensions and generalizations. The emphasis here is on the implications of the use of similar modules, rather than on the particular characteristics of these devices.

1.2 Small ships electronics systems

The subject of this report has direct bearing on the design of electronic systems for small ships because of the extensive use of digital information systems deemed necessary to fulfill the unique small-ship requirements.

The general requirements of the instrumentation of small ships include nearly all the functions found elsewhere in the Navy, but they also include tighter restrictions. The systems must be effectively maintenance-free over reasonable-length mission, the equipment must be even smaller, lighter and more environment-proof than ordinary, and the functions of the system must be implemented in such a way as to require minimal manpower for operation. In addition, the operation of these vessels must include all of the types of tactical functions which are required aboard their big sisters, and must be extremely responsive to remotely commanded control.

Necessarily, the solutions to the small ships problems will include solutions to problems in common with other ships. Thus the attributes not unique to small ships will lead to results which benefit others, as well.

The solutions to the problems of creating complete systems in smaller packages will give better unified systems, designed with intelligent use of more highly automated techniques. The solutions to the lower manpower requirement must be better integration among subsystems, better cooperation among their respective functions, and more effective approaches to the operator interface. The solution to the problem of widely diverse, changing sets of mission requirements can be the use of faster techniques for producing effective designs, coupled with a wide-ranging approach which provides the capability of easily reconfiguring the system among different ships, or between different missions of the same ship. This report includes discussion of the design approaches necessary for producing such effective, adaptable organizations.

1.3 Soft, firm and hard wares

As described in Section 2.2, a statement of the steps to be performed with a particular set of equipment in order to compute a certain result is a program. Depending on what equipment is being used, this program may take on one of several forms. If the sequential order of the program steps and the steps themselves are completely expressed in terms of the equipment itself, they comprise a hardware program. Thus, hardware connotes a fixed organization; a sequential/combinatorial logic circuit; "hardwired" connections.

If the program consists of a structure of instructions fed to an instruction processor, and if the instructions are encoded in some easily changed medium than it is a software program. Specifically, a traditional computer program is software. It is a collection of statements written for a specific computer or written in a fixed language for an unspecified member of a collection of computers, but the program organization is "soft"; it is easily changed by the programmer. The computer program is written on paper, punched in paper cards or tape, or coded in electromagnetic field patterns on transmission lines, magnetic storage surfaces, ferrite cores or multistable electronic circuits. The operations of the programs derive from the sequence of statements written in the program language, interacting with the progress (the state) of the program. If the program is accessible to computation in much the same way as the data, the program may be self-modifying. In contrast with a hardware program, a software program is generally a more pliable, transient form. It is more exclusively concerned with the pattern of the computation itself, and less with the mechanical details of realization.

Some programs are not conveniently defined as either hardware or software. The term firmware program has two simultaneous connotations. One is that the program has some similarities both to hardware and to software. For example, it might be expressed as an electronic circuit but contains stored patterns which are decoded into operations and which vary with the results of the operations. The second implied meaning of firmware is that its degree of permanence is intermediate to two extremes; the program structure is neither hard nor soft, but firm. The microprogram in an IBM 360/30 resides in a card capacitor read-only storage. Microprograms in this machine are not self-modifying but the cards are easily changed. Such a machine might, by changing microprograms, be used as a standard 360, as an emulator, and as a machine for direct execution of a higher-level language, all in the space of a few hours. Particularly in the case of microprogramming, the term firmware is an invention required by the existence of programs which are neither clearly hardware nor clearly software.

The distinction between hardware and software has two useful purposes. First, it is useful as a short abbreviated description as used by technically competent people. In any engineering problem, the hardware is the physical equipment; the software is the organization and the ideas. This is still true in data processing problems, but when one is concerned with the organization and programming of the physical system (the "software" of the "hardware") or the representation, coding, storing and transmission of the instruction stream (the "hardware" of the "software"), it becomes obvious that the distinction is neither precise or adequate. The terms are useful, to people who understand their shortcomings, as qualitative and sometimes transparent indicators.

Second, terms hardware and software are popular in external descriptions of technical activities. The terms provoke vivid though sometimes misleading images, and they provide attractive handles for use by someone who wants to pretend he understands what is happening, or by someone who tries to manage or guide an effort without attaining such an understanding.

The concern of this report is not with hardware or with software, but with organizations, operations, and the design and preparation of programs. In the next several sections, various aspects of this process of systems design are examined for the particular case of digital information systems. The most important theme throughout the discussion is the manner of dealing with programs. A generalization of the attitude is, that program development must be carried out in a consistent and efficient manner without making a priori distinctions between hardware and software. An effort to design a large system which starts by making large-scale "hardware/software tradeoffs" is inviting premature, unnecessary restrictions too early in the design process. More meaningful large-scale choices are the organizational aspects such as centralized vs. dispersed, serial vs. parallel, synchronous vs. asynchronous, etc. Much later in the program design come the decision, for example, as to whether a program module should be implemented in a traditional vs. innovative computing structure (read software for a general purpose computer vs. microprogrammed devices vs. hardwired circuits, if you insist). An effort which proceeds in a balanced manner can lead to choices which are more soundly based on pertinent consideration of performance and resources, and which can easily be changed by modular replacement in the event of a change in requirements or a change in the available technological base.

2. Levels of organization

A good system is a well-organized collection of properly chosen components. A modular system is a collection of relatively few component modules, each of which is relatively complex. But each module is composed of interrelated items, which themselves may be complexes of others, etc. For any given design, the components are the smallest invariant units of the design. The systems are the largest design units, and subsystems are convenient intermediate-level complexes. But one man's components may be another man's systems, in some different situation. We are led to conclude that a complex system design is best described at a number of different levels.

2.1 Levels of design

Bell and Newell (4) define the levels in the hierarchy of digital computer structures largely from the basis of considering the different activities of different technical practitioners. The "institutionalized positions" of logic designers and circuit designers are used as evidence for the existence of distinct levels. Their highest level (the so-called PMS level) has computers as structures and processors, memories etc as components. The next or programming level sees programs as made of component instructions, operators, etc. The three logic design levels are:

the register transfer level - arithmetic units made from registers, controls, data operators;

the sequential switching circuit level - counters made from flipflops, latches, oneshots;

the combinatorial switching circuit level - encoders, selectors, iterative nets made from logic gates.

The lowest level they consider is the circuit level where example systems (circuits) are amplifiers, clocks and gates and where the components include relays, transistors, resistors, diodes, delays.

One difficulty with defining a level of design as the concern of an identifiable group of people is that this produces a passive view of past practice. A more appropriate criterion might be to identify a level of design with a design language: then

"A system level... is characterized by a distinct language for representing the system (that is, the components, modes of combination and laws of behavior). These distinct languages reflect special properties of the types of components and of the way they combine.... The fact that the languages are highly distinct makes it possible to be confident about the existence of different levels." [4, p.4]

This method of identifying the various levels of system design allows us to identify the most recently emerged levels, but it leads to a significant difficulty. Whenever it is difficult to decide whether two languages are "highly distinct", it is also difficult to decide whether they define different levels. Thus there is no effective procedure, even in principle, for counting the number of distinct levels of system design.

The number of levels, and thus the extent or depth of the levels, are not precisely known. This is one reason for introducing the supplemental identification of a level with the activity of a man or group of men; it solidifies the general identification of major levels, without concentrating on the the boundaries and without worrying about the fringes.

This view introduces a new notion: the span (depth) of a level is commensurate with the comprehension of a human being. That is, one historical reason for designing a large system in successive stages has been that the human designer has a certain limit to the range of detailed consideration which he can handle effectively. If the design process is to be automated, it might well be done in smaller steps (for the machines are notoriously inept at handling the intuitive associations which a designer employs) or larger (if the machine can cooperate by performing routine tasks and interface well with the creative skills of the human). The number and depth of the design steps is not precisely determined now, and may change drastically in the future.

For any definition of a level of system design, however, the design language is the key to the existence of a level. The language defines the level; is the tool for designing at that level; expresses the components and systems of the level; and provides the documentation for design at that level. The lowest-level, irreducible units of a design are the primitives (words) of the language; the system structures designed at that level are the utterances (sentences) of the language. Preparing a design at a given level means writing a statement in the language of the level. The process of designing an entire system becomes a process of carefully translating statements in one higher-level language to successively lower levels.

2.2. Programs

Information processing machines are action-oriented; they perform operations on their inputs and produce outputs. With rare exceptions, they are not goal-oriented; they do not actively seek a "solution". The driving force connecting a problem with its solution in terms of action-oriented processes

is the organization of the processes. Operating processor units are connected, sequenced, and in general, controlled, according to some predesigned scheme in an attempt to solve the problem. We call the specification of a desired pattern of operations an algorithm. The realization of the algorithm in terms of available mechanized operations is a program.

Although useful in the present context, this distinction is not universally accepted. One definition of an algorithm, which derives from work by A.A.

Markov, requires that it be a specification of a sequence of operations which is:

- a. General: it operates correctly for any input in a specified domain.
- b. Conclusive: it uses a finite number of deterministic steps and always terminates.
- c. Definite: The specifications and the required operations are unambiguous and "universally comprehensible", [8].

Unfortunately, given a purported algorithm it can be very difficult if not impossible to determine whether or not it satisfies the three requirements. Algorithms must be written in some language; the choice of a language influences how close the statement of the algorithm comes to meeting these three requirements. Use of a natural language such as English, along with standard mathematical notation, makes the statement more widely comprehensible, and sometimes quite general, but can leave it ambiguous. Proving a statement correct has so far been accomplished only for restricted classes of problem structures for small problems.

Practically speaking, however, algorithms must be written for distribution, discussion and implementation. The Communications of the Association for Computing Machinery publishes algorithms. Their solution is to accept statements of algorithms written in one of a small number of widely known computer programming languages, such as ALGOL 68, or FORTRAN. In one sense, this avoids

the problem of writing the algorithm and substitutes a computer program. Publishing a program has several advantages:

- a. It is universally comprehensible to the large set of people who can read the language;
- b. It is unambiguous to the degree the programming language is;
- c. It is subject to estimates of its generality (also published are certifications, statements attesting to a partial audit, or verification that the program operates correctly when run with some data by some computer system); but conclusive proof of the property that it always terminates correctly for all inputs in the domain is generally absent.

3. Program Preparation

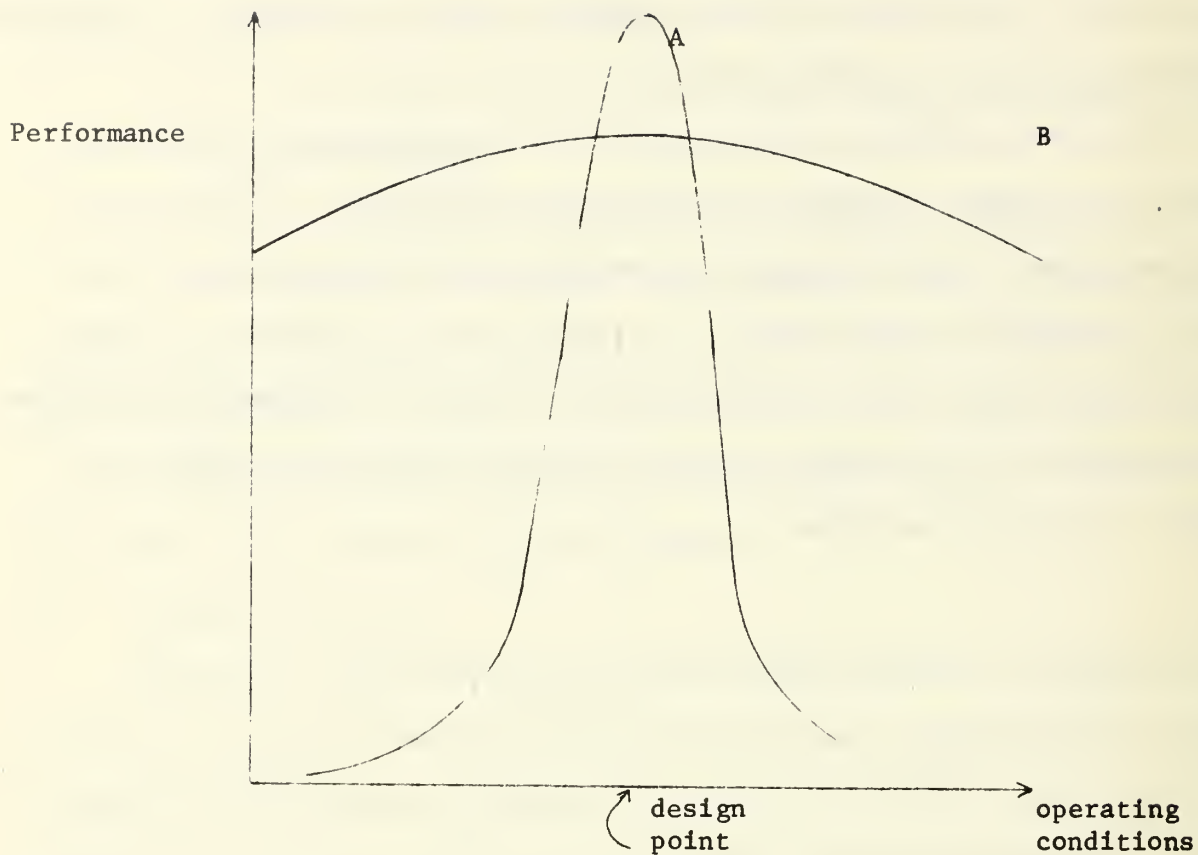
Satisfying a functional specification means preparing a program to perform the algorithm inherent in the specification. Identification of this algorithm is often the most difficult step in this problem; even if the "user" thinks he knows what he wants, he will usually change his mind before the final product is produced. A competent designer, knowing he is faced with incomplete and changing requirements, has two defenses.

First, he builds systems which not only perform well in the specified environment but which also perform moderately well over a reasonable range of conditions. Such a design, which is able to stand up well under a broad range of conditions. Such a design, which is able to stand up well under a broad range of conditions, is technically known as a robust design; the concept is illustrated in Figure 1.

Second, the designer prepares to meet changing requirements by building a repertoire of techniques to respond quickly. This repertoire includes carrying out each necessary design step in a somewhat generalized manner, so that changes in performance requirements or operating conditions can be

propagated through the design levels from top to bottom with a minimum of extra effort. But probably the most important part of the designer's repertoire is the ability to use modularization.

Of course, systems are constructed as an interconnection of discrete modules for a number of reasons, including task management, reliability and fault isolation. Modular design is stressed in this section, however, as a means for achieving systems which are relatively easy to redesign. A change in the specifications for a modular design can often be implemented by changing the parameters of a few modules and adjusting the interfaces, without disturbing the rest of the design.



- A. "Optimized" system; best performance at the design point, but poor performance at nearby, reasonable conditions.
- B. Robust system; performs well under a variety of conditions.

Figure 1. A Cartoon Showing a Robust Design

The design problem in general is to implement a required function (a component of a higher-level system) as a system of a specified set of lower-level components (themselves, perhaps, systems of still lower-level components) according to a specified technique of description. Thus, for example, a circuit design problem might be to design the logical inter-connection pattern of a JK flipflop composed of resistors, capacitors, diodes and transistors according to the transistor-transistor logic (TTL) technique.

3. 1 Traditional software generation

As background and to provide a contrast with the attitude in the remaining sections of this report, it is worthwhile to briefly consider some aspects of computer program generation. Traditional software generation here refers to the writing of computer programs in a procedural compiler or assembler language, and the subsequent processing (conversion) of these programs to machine code for execution in a general-purpose computer. No attempt is made to survey the entire field. Instead, a few comments are made about concepts which will probably be found in a Navy program generation center within a few years.

One concern which will continue to be important is the topic of proper documentation for a program. The Navy has recognized that in order to monitor the production of a program, in order to maintain the product, and even in order to properly use the product there must be adequate documentation. The reason documentation has notoriously been inadequate and behind schedule is that it has been considered as something separate from the design. This attitude toward documentation is perhaps the only distinctive difference between "hardware" and "software" designers. For mechanical hardware, all of the design is recorded on paper -- drawings, schematics, analysis. For computer programming where the code is the product, the softness itself contributes to its disparity with other documentation. Sufficient and proper documentation of programs will not be achieved until coding the program into a programming language becomes merely another design step, and not so much an artistic creation. Then the design step which we now know as coding will contribute to, not distract from, the program documentation.

The practice of designing a large system as a collection of smaller tasks is universally accepted as necessary just to complete the job. The fundamental benefits of modularity (in specification, design, production, maintenance and update activities) accrue only when the modules are carefully chosen with respect to their functions and interfaces. Unfortunately, it appears that near-future practice in Navy command and control projects will be dictated by a heritage from NTDS and the influence of a few major manufacturers' organizations. Modules, by way of NTDS practice and collateral "standards", seem to be predefined in terms of operations which derive directly from machine and language artifacts, e.g. subroutine calls. Interfaces often occur in the form of control blocks or parameter lists. Consequently, designers of each module are encouraged to acquire and use knowledge of details (such as the bit structure of a control word) which must be handled identically by several different modules. In some primitive forms of data base design, the programming details of each module which accesses data are largely dictated by details of the specification of the data base. The severe problem caused by this widespread interdependency occurs because the design process is not a unidirectional, one-pass activity. Even in the initial production of a program, changes occur as the design progresses. During and after the production of the first version of the system, analysis, experimentation and measurement results are reflected in changes in the amount and format of data which **must** be stored in these common areas. If the detailed structure of the common control block or the data base is known and used by the designers of several modules, all of them reflect every common block change in their own designs. Designing a system around common blocks thus seems like a method to maximize the interference among modules rather than minimize it. An alternative approach which minimizes the knowledge needed by one module designer about internal details of

modules, and therefore minimizes the interference of design changes, is being developed [9, 10], but it appears that widespread adoption of such principles in program generation centers will be long delayed.

Another topic which should be given more attention, but which will probably not appear as changed programming practice in near-future program generation centers, is software reliability. Present program generation center attitudes seem to assume that a properly written and tested program is perfect. Perhaps it is, but software troubles show up later for two reasons. One is that a program tested at the Center may encounter in the field inputs which were not in the tests. Another reason is the volatility of the code; transient signals in the computing equipment and mechanical errors in the program transfer equipment can change the program. Just as mechanical designers know that simplicity in the basic design enhances the reliability of their product, so computer programmers must believe that unnecessarily complicated code leads to programs which are difficult to read, test and debug and maintain. If any complexity is to be added, it must be carefully chosen as protective redundancy -- built-in data and program checks for reasonableness, consistency, bounds, etc.[15].

A final comment about the near-term trend of computer programming concerns both standardization and languages. Standards are vital in a mature field of design in order to insure compatibility among organizational units which are designed separately. Languages are the tools of design, and the use of standard tools can help in producing standard products. But to "standardize" all of the Navy's command and control programs by requiring the use of an inappropriate and antiquated language (FORTRAN seems to be an attractive candidate to some people) is to require inappropriate solutions and to legislate inefficiency.

Indeed, if such attempts are carried too far, they seem to be efforts to use "standards" or pro forma solutions, and to abdicate the basic responsibility of design, which is to select, configure and interconnect appropriate elements in the manner which is most appropriate for the application.

A recent tendency related to standards and languages is the topic, recurrent in literature concerned with Navy command and control systems, of "high level" languages. This is partly an over-reaction against the primitive assembly language - like constructs still found in CMS and CMS-2 programs. The best payoff, in terms of portability of programs, will not be found by legislating the universal use of "high level" languages in hopes that this will guarantee machine-independent code (it won't), but rather by allowing more freedom in the choice of language and by encouraging the consideration of portability in the entire system design. If computer program generation were a mature design process and less an art, there would not be a need to require a certain language. The designer would choose the language appropriate for the task at hand. As it is now, the degree of machine independence is much less a function of the language used than of the way in which design structural information is or is not employed. Programming in a "high level language" does not imply "high level thinking"; in fact, many unrealistic dreams can be punctured by a little "low level", informed reasoning.

3.2 Life cycle considerations

As mentioned in the preceding section, the activities involved in producing today's Navy software systems often seem to assume that the job is finished when the program is written. This is patently false. Even so, the present mode is to contract for the procurement of a deliverable program without sufficient specifications or tests for its future performance characteristics.

The procurement of any system goes through a number of phases in its life. Successful performance in one of these phases may be jeopardized by failure during an earlier phase to consider the future. Tradeoffs must consider the entire system lifecost, not just design or production.

Listed below are several successive phases of the life of an arbitrary design. The life of a system is shown here as indefinite, in the sense that many of the original components will be replaced, and the system is capable of continued evolution. The reason for listing these phases in this report is to abstract some system characteristics which are independent of the hardware/software nature of the implementation. Hopefully, system designers of today can read such a list and find it equally familiar and equally pertinent, whether they are thinking of software systems of program modules, mechanical systems, or logic systems of hardware, firmware or whatever modules. Hopefully, too, a system designer of the near future could read the same list and find it quite familiar. He will know that what is implied is any arbitrary mixture of hardware and software and hybrids; the choice will be made (and can be changed) at the level of a specific subsystem. He will probably not even care that at some time in the past people considered software system design and hardware system design as different activities.

The creation of a system begins with its specification. The first design step is to develop performance requirements or statements of how the system is to relate to the world, how it is to be used, and to reflect that use into functional specifications, a statement of the functions which must be implemented in order to satisfy these needs. Unfortunately, this phase of the design is sometimes the most difficult. Especially in the case of new information processing systems, it is difficult to predict the user requirements for a

capability that has not existed before. A good example is the public experience with timesharing systems. No designer knew the wide diversity of uses which would be asked of such systems until the first ones were built. No one knew the performance demands until there were users. This difficulty is still present; see for example the decisions being made about page replacement algorithms for the AADC, without knowledge of what the actual performance requirements (in terms of the dynamics of AADC program working sets) will be [11].

Design of the system proceeds from functional specification through various phases of organizational or structural description to design of the complete system as interconnections of cooperating processes. Each process might be a component procured from another source or it might be a subsystem which itself is subject to design. Other sections of this report have more to say about the methods and tools of this series of modular design steps so here we will only add a few comments. If the complete system is to have any of the desirable attributes of reliability, maintainability, portability; if it is to be subject to verification and test, monitoring and measurement, redesign and upgrade, the capabilities for providing the means must be built into the design. Specifications for providing the measurement hooks, for detecting, locating and isolating errors, for module replacement must be built into not only the equipment but the design information. Ideally, the successive refinements of the design become the basic documents of the system.

As modules and components are specified more precisely, the design steps become experimental production steps. The only difference worth noting here is that gradually the system begins to exhibit some of the originally required functional specifications and some of the capabilities in the original performance specifications. A vital point here is the necessity for testing each step of

the gradual combination of modules to verify the validity and performance of the combination.

If the organization of the system is sufficiently modular and if the interfaces are well designed, the system is amenable to successive updating cycles starting with the experimental production phase. In fact, for large first-time systems, applying several rounds of updates to portions of the experimental production model seems to be the most common way to produce an operational systems.

Two more phases of activity concerning the life of the system, training and maintenance, must be mentioned. These are the activities which use the features of design which are mentioned above, but which in practice are often ignored as being incidental to the primary purpose. The functional subdivision and organization of the system can be exploited in training people to use, operate and maintain it. The modular nature of the design can be an aid to isolating and localizing difficulties which need maintenance. The designed-in protective redundancy can minimize the effect of errors, and designed-in test measurement facilities can help in quick repair.

3.3 Means of program preparation

If creating a design to meet a given set of specifications means preparing a program to perform the desired function, then it can be seen that a fundamental asset to a designer is a set of tools and techniques to help him prepare programs. As these tools become more comprehensive and more thoroughly mechanized, they become elements of design automation. "Design automation is ... a way to reduce design time and overall product cost, ... a [a way to provide] documenting, ... analysis ... and synthesis functions..." [12].

The present situation in the field of automation of digital (hardware) systems design sees computers being used in several ways. As described in a

recent survey article [5] systems are used for simulation, synthesis or translation, partitioning, placement interconnection and fault test generation. Simulation capabilities are concentrated at the gross system level, the gate level or the circuit level; relatively little attention seems to have been paid to intermediate level simulation (exceptions are more carefully examined in Section 4 of this report). Aside from simulation, other functions of equipment design automation seem to be mainly concerned with low level organizations such as gates or circuits. An example is the class of partitioning problems; given a system of low level components (i.e. a conventional gate level logic design containing thousands of gates), separate the system into several parts. Optimization is attempted with respect to number of connections, number of parts, number of types of different parts, etc. The goal of this class of problems is similar to that of several other efforts; to specify system building blocks. The difference is that the partitioning problem starts with a particular low-level design and groups components. Functional modularization efforts, in contrast, try to identify modules which perform a certain action which can be found in many different designs, and to design systems using these modules.

4. Functional Program Modules (FPMs)

4.1 Design as configuration

One of the characteristics of the classification, "small ships", is its diversity; the types of operational requirements vary greatly. An integrated approach to the preparation of information systems for small ships must face the problem of assembling systems differently tuned to the different requirements.

One solution to the problem is to design a general purpose system which has every one of the capabilities required by each of the ships for each of its possible missions. Of course, previous design experience has taught us that

the overhead required to provide all these capabilities, to integrate them, and to manage them, costs dearly in terms of the physical parameters of the system, such as weight, size, power, and the operational parameters such as maintenance, availability, efficiency and speed of processing.

Another characteristic of this class of ships is their small size; a large general-purpose information system here is competing for scarcer space, receives fewer inputs, and feeds fewer outputs than its counterparts in larger vessels.

For these reasons it appears obvious that the small ships electronics effort requires special-purpose configurations of functional components. Apparently the main task is, given a specification of the operational functions to be performed and given a physical ship configuration of sensors, weapons and control, to configure the signal processing functions necessary to accomplish the operations in the best possible way. Several auxiliary problems immediately arise; preparing the operational specifications, deriving the functional specifications, discovering the pertinent sensor, weapon and control characteristics. But the configuration problem, or the design problem, is particularly interesting.

The design problem is too big. It is beyond the present techniques to proceed in one step from operational specifications to wiring diagrams, physical configurations and machine language computer programs which accomplish the goal. This design step is too long, for the same reason that ABM opponents say the system cannot work; the design procedure cannot today cross the many levels of successively finer detail in a single step. Instead, the process proceeds from one level to a lower and arrives at a detailed design after a series of steps.

An idealized design procedure in the technology of the 1950's or 1960's for example, might start with a loose statement of operational functions of a platform. This statement, when compared against available technology, might

result in functional specification of a ship's hull, propulsion, personnel, sensors, weapons, communications, and information processing. Each of these areas would then be carried to the next lower level. The information processing level, for example, might be carried to the level of specifying a configuration of computer systems, preprocessors, convertors, modems and mechanical and human interfaces. Another design step then might be required from the computer system to the specification of processors, memories, controls, channels and peripherals. The design of the processor might be accomplished at the level of data operators, registers, busses, decoders, controls and interfaces. The next lowest design level might be the logical design of a subsystem such as an arithmetic-logical unit in terms of combinational and sequential logic elements such as gates and flip-flops. Finally, the circuit design implements a flip-flop, for example, as a configuration of resistors, transistors, capacitors... A resistor is designed too, but this is carrying things a little too far, even for the era of 1950.

The multilayered design process as described above is idealized to make a point. No mention was made of the physical, operational and political constraints on the design step between any two levels--these constraints are the factors which make the design step a design problem. The process was described as a straightforward, top-down, step by step procedure. In reality, each step loops back to previous higher levels and iteratively interacts with the higher level design problem. Finally, many of these design steps are not required for a given project. Off-the-shelf components are used at some level. The information processing system for a given ship, for example, may be designed as a configuration of available computers with available interfaces to already stocked sensor, communication, fire control and navigation subsystems, using previously procured displays and consoles. These deviations from reality in the above idealized picture, however, do not destroy the main point--that the design of a system to meet operational requirements is a sequence of steps, each configuring an element of a higher level from components at a lower level.

4.1.1 Standards as design

Every designer uses standards such as the industry-wide standard sets of functional parameters, physical dimensions or calling sequences. Next to the principle of assembly-line manufacturing, the wide use of standard components is probably the most important factor in this nation's industrial successes. Information processing designs are evolving standards of their own.

The designer also relies on standard design techniques. Each industry and each design group has certain common conventions which make design documentation more easily comprehended across the group. Thus, each of the designers involved can readily understand and contribute to the design. The main benefit in standard design formats and conventions for structural drawings, flow charts, logic diagrams and programs is that the information recorded tends to be concise yet unambiguous. A second benefit is that required formats make it easier to insure that all the necessary design criteria are satisfied and sufficiently documented.

Standards in use among several different groups contribute to compatibility of their separate products, but do not insure compatibility. The improper use of standards can interfere with the design efforts. If inappropriate "standards" are externally enforced they can require inappropriate design measures. For example, TADSTAND 2, 8 Nov 1971, specifies that the standard for tactical data system program documentation is given by Weapons Specification WS-8506. WS-8506, for example, partially defines the structure of the program by defining such pervasive terms as program, subprogram task, common data base, constant, etc. and requires documentation in these terms. A design of a system which is based on a different set of elements (a multiprogramming, virtual machine concept, for example) might not conveniently fit this format. Either the design must be

recreated for the purpose of documentation alone, or (the path of least resistance) the preferred program structure will be abandoned in favor of a "standard" approach which allows documentation as part of the design process. Misuse of inappropriate standards can be a substitute for good design.

4.1.2 Design/configuration aids

A major thrust of research in signal processing systems architecture has been to make the design step easier, faster and more highly automated. The historical progress of technology throughout the world has been the increasing ability to produce increasingly more complicated designs by successively incorporating more and more of the knowledge and techniques from previous designs.

Development of the effective techniques for the design step at any given level requires three things: identification of the levels of organization involved in the design, development of a language for expressing the design, and development of techniques for translating the higher-level requirements of the design to lower-level specifications which realize the design. All three of these are usually accomplished gradually for a given design level.

The decision as to which components are taken as elementary units for a design level, and which are to be considered elements (systems) of a higher level, is generally an evolutionary process. Subsystems which are repeatedly designed and manufactured with essentially similar functions become, over a period of time, components for design at a higher level. Thus flipflops become components of logic designs, registers become units of arithmetic designs and arithmetic logic units become units of central processor designs. A subsystem becomes a component when its design is established, when it finds wide application in a number of different places, when it becomes available as a unit

from some source, but most importantly when a designer finds it convenient to use it as a unit in larger designs for purposes of comprehension, documentation and management.

The language of a design level also grows gradually, being composed partly of notation from the lower, more detailed level and partly of generic forms common to many design levels, such as signal flow block diagrams and flow-charts. However it arises, the language is the vehicle for describing two vital parts of the design: the structure of the system (its logic inter-connection) and the function of the system (its logical operation).

Techniques for translating system design requirements from one level of description into realizations in terms of organization of components at the next lower level of description gradually arise as the components become identified and as the language definition evolves. Many examples come to mind from the field of computer software design. Assemblers developed as the translation tools from the assembly language level of labels, opcodes, addresses and modifiers to the machine language level of bit patterns (fields) in computer words. Macro assemblers developed as the translators of functional patterns to assembly language code. Compilers arose as the translators from imperative statements (e.g. $X = A + B - 2.*Y$) to lower levels of macros, assembly language or machine code. Linkage editors and loaders developed as the tools for translating the job control language for specifying a system of interconnected programs into the actual package of executable code, constructed from modules developed at the next lower level of program design. Operating systems now have developed to the point where they implement the actions of jobs by manipulating the physical computer system, allowing the applications programmer to use convenient, idealized, and sometimes imaginary memory, operations, and communications.

4.2 Register Transfer Modules (RTMs)

One approach to the design problem at the register-transfer level is embodied in a set of design units known as Register-Transfer Module (RTMs)^R. The effective use of these modules requires an understanding of the register-transfer design level, an understanding of flowcharting language statements, rules and components, and a cautious consideration of the costs in equipment and efficiency when the technique is applied to systems which are too large or too small.

4.2.1 Origin of RTMs

RTMs are a realization of the register-transfer level of design as described by Bell and Newell [4,p3-11]. It is that level which is concerned with design of systems such as arithmetic units from components such as registers, transfers, controls and data operators. Another approach to a quite similar level has been reported by Chu [6,7], but this work concentrates on the use of an intermediate language called Computer Design Language (CDL). CDL is essentially a language for simulating sequential digital computers whose structures are similar to conventional second or third-generation general purpose computers. Certain features of the language make it difficult to apply to innovative designs involving such features as a hierarchy of nested levels or parallel or subroutining organizations. The RTM approach has two important differences; it is a general approach to design at the register-transfer level, and it leads not just to simulation but to hardware realization.

The basic components of the RTM method are, as described by Bell and others [1,2], realizations of components derived from the PMS level of computing structure descriptions [4, esp. pgs. 10-20]. The operations (funtions) performed

^RTrademark of the Digital Equipment Corporation

by the use of these modules are of course register transfers, described mostly in terms of action sequences in a nearly universally accepted version of the ISP language [4, esp. pgs. 22-36, and g], viz:

$$A \leftarrow A+B$$

By concentrating on the register-transfer level of design rather than on the structures to be built from this level, the RTM technique has acquired a wide generality in the possible type of application.

As described by Bell and Newell, digital systems have memory (M), data operators (D), transducers (T) and controls (K). Other types of components are involved, but these are enough for examples. Different forms of these components are described in the PMS notation by strings of abbreviations and numbers, arranged to be quite sensible by someone with digital system design experience. The influence of the PMS descriptive system is shown in the names of the following examples of elementary functional RTMs:

Ksimple (or Ks), simple control. Upon receiving a signal from a logically preceding module (another K), the Ks issues a signal to evoke some specific active such as a data operation and transfer, and when the Ks receives notice of the completion of that action it passes a signal to the next element (K), which, typically, controls the next step in the program.

Kdecision (Kd). According to the state of some logical variable in the system, the Kd passes the control sequence to one or other of two following sequences, implementing a program branch.

DMgpa (A, B). A general purpose arithmetic data operator with (16-bit) memory registers A and B. Performs elementary fixed point binary addition and subtraction and several logical operations such as shift and logical AND.

Kbus. A bus control. Provides a memory register to latch the content

of a signal bus, provides a means for controlling bus operation and for sensing simple conditions on the bus.

Many more modules are possible; some have been implemented by DEC and others have been described and used in other descriptions. But these few form the nucleus of a capability for programming.

Figure 2 shows a program, complete except for input and output, to implement a nearly trivial example algorithm:

$$\text{form } S = \sum_{i=1}^N i \text{ by addition.}$$

A similarly sequenced FORTRAN-like program segment for this algorithm is:

```
S=0
READ N
100 IF (N.EQ.0) GO TO 200
S=S+N
N=N-1
GO TO 100
200 WRITE N
```

Aside from possibly a few flowchart conventions, Figure 2 completely specifies the system. Automatic techniques such as those described later under the heading "PDP-16" can be used to provide a description of the hardware in more conventional terms such as racks, circuit cards, power supplies, cables, wiring lists, etc.

Instead of defining the RTM design language here, we will let Figure 2 stand as an example. The design language for the most part consists of flowchart symbols interconnected to show control sequences. As distinct from other flowcharting schemes, however, statements inside the RTM flow boxes are restricted

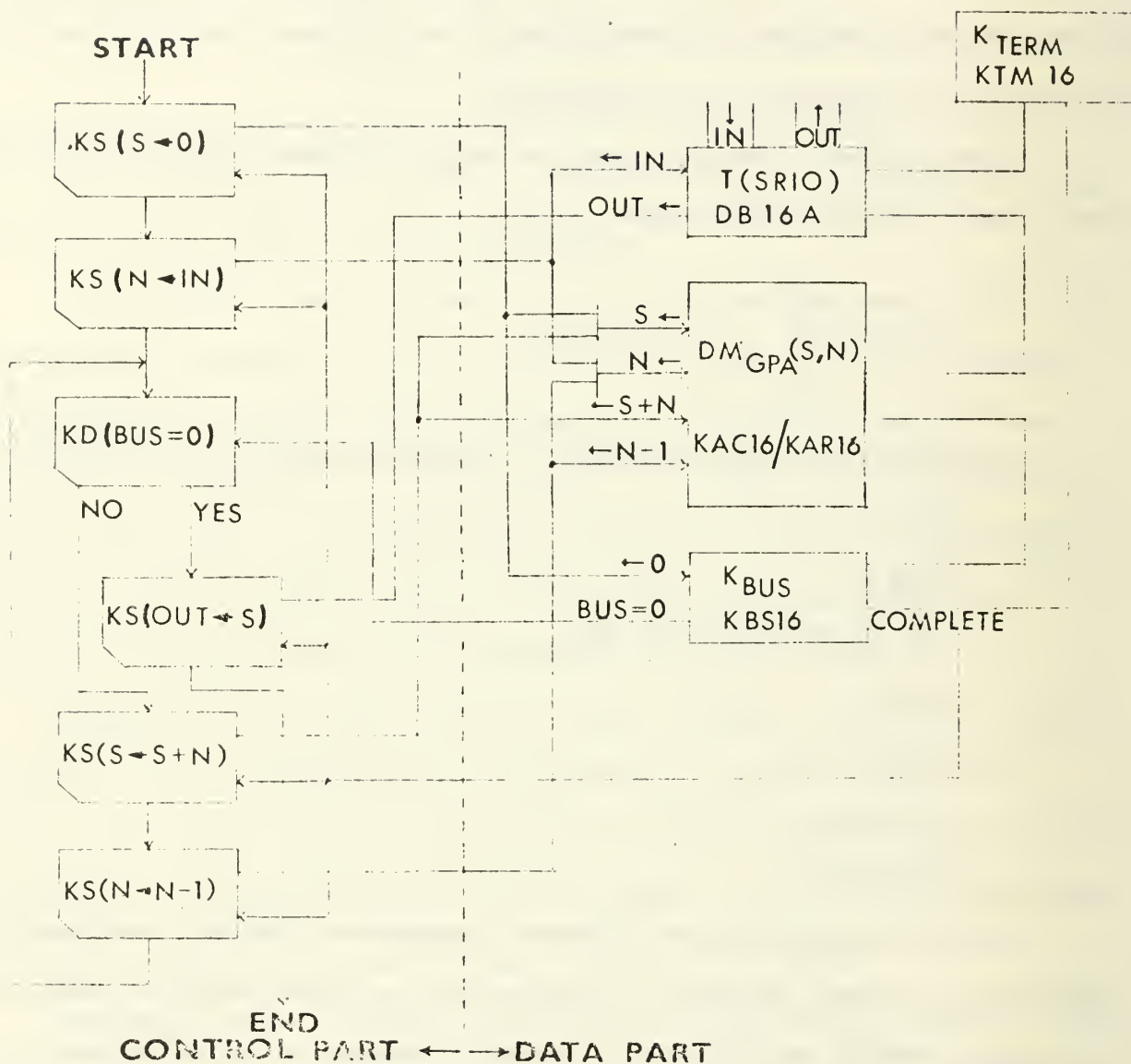


FIGURE 2 AN EXAMPLE RYM PROGRAM

to simple expression forms at the register transfer level. A typical example is the data operation type:

$$R_1 \longleftarrow R_2 \text{ op } R_3$$

R_2 (optional) and R_3 are source data registers, op is an operation specification, and R_1 is a destination data register. Such statements are reminiscent of the elementary "triples" of the syntax analysis portion of traditional compilers.

A striking feature of a typical RTM design is the natural division between the control part and the data part. On the left of Figure 2 are the interconnected control modules. The module types (Ks and Kd) and their connection pattern characterize the structure of the program. On the right, the data operators and bus(es) characterize the implementation of the operations of the program. Between the two parts are a large number of interconnections.

Because the control part, with data operations specified inside the flow boxes, almost completely determines the connections to the data part, and because the data part is essentially the same for most simple examples, many of the examples of RTM designs given elsewhere show only the control part.

4.2.2 PDP-16

The Digital Equipment Corporation markets a set of RTMs bundled with design services (personnel and PDP-10 programs) and collectively called the PDP-16 [12]. The marketing effort appears to concentrate on a part of the activity served by their Control Products group. The PDP-16 method is presented as appropriate for design of fixed-program industrial control systems which are neither too simple (<20 bits of memory, <20 control steps) nor too complex (>1K words of memory, >250 different control steps). Smaller systems are described as appropriate for design by "standard" logic modules, and more

complex systems are characterized as appropriate for the use of minicomputers. DEC has strong vested marketing interest in both of these alternative design methods.

A most attractive part of the PDP-16 system, from the point of view of a customer, is the idea that designs are implemented directly from the flowcharts. The flowchart elements represent the actual hardware which will be used for sequential program control and for data operations. From a flow chart provided by the customer (DEC gladly provides this, too, for a price), the design aids produce listings for module insertion, wire wrapping, checkout, installation, and maintenance. Card decks or paper tapes for automatic wire wrapping machines can be obtained directly.

4.2.3 RTM Simulation

Once the basic feasibility of implementing has been established by building and testing some basic modules, and by using them in some sample designs, attention turns to alternate and expanded sets of modules. As explained in the previous section, the RTM modules which have been built are mostly based on the descriptive system for digital computers. The PMS language is a generalized attempt to describe the structure of any general-purpose digital computer at the highest level. The RTM modules are described by PMS notation, and seem to have been selected to correspond in large part to the major units of the PMS scheme. Deviations and modifications from this correspondence seem to have resulted from consideration of the packaging of the physical modules themselves. When attention is turned to design of dedicated special-purpose signal processors rather than general-purpose computers (and RTMs are recommended only for the former), it would seem that perhaps a different set of modules would be more effective. For trial designs and initial evaluation of new choices for modules, simulation of their performance is preferred over actual physical construction. In preliminary phases,

simulation, when soundly based, provides quick, easy results and the convenience of flexibility in changing modules and their characteristics. Recent simulations [13] have included the following:

Basic modules

Ksimple

Kbus

Ktest

GPA

Arithmetic modules

Adder

Ktimes

Kdivide

Kbranch

Kboolean

Floating point modules

Fadd

Fsub

Fdivide

Ftimes

Trigonometric

Sinang

Arcsin

Angtan

The "basic" module simulations established the soundness of the approach. The others were mostly extensions to provide useful functional units in a design for a particular problem ("triangulation"; computation of the lead

angle of intercept for a weapon) appropriate to a mobile defensive weapon platform, and for use in a demonstration of the feasibility of implementing with RTMs the fundamental FORTRAN computation/control structures.

4.2.4 Other Examples

Briefly described below are some of the applications to which the RTM design technique has been fitted. In each case the problem is briefly described, and a reference for further explanation is cited.

1. Sum of integers. As in previous Figure 2. See also [2] p. 15, Figure 3 and [13], p. 20.
2. Multiplication by the Booth algorithm forms a 16-bit product from two eight-bit inputs. [2], p. 15 and figures 4 and 5. Illustrates use of two busses and the use of subroutines.
3. A remote A/D sampling unit interfaced to a serial synchronous modem, [2], p. 23 and Figure 7.
4. 16 x 16 multiplication [13], p. 32.
5. Interfaces: from PDP-16 to
 - a. PDP-11
 - b. PDP-8e
 - c. A 1K x 16 bit core memory
 - d. A 32 x 16 diode ROMreference: [14], pp. 148-155.
6. An A/D preprocessor for a central processor. [14], p.29.
7. A magnetic tape code translator [DEC tabloid flyer, p.5].
8. Small general-purpose computers. References: [2], p. 23; [1] p. 498-499. One was a computer very similar to the PDP-8:

"...an RTM diagram for a small stored-program computer that was initially constructed as an application experiment to demonstrate the feasibility of the modules and to investigate systems problems. The process of specifying the machine took approximately two hours (with three people). The computer was wired and, aside from minor system/circuit problems (for which the experiment was designed), the computer operated essentially when power was applied, since there were no logic errors." [1], p. 498.

Other computers have been designed to emulate the PDP-8 and the Data General NOVA.

9. Functional modules

Within the basic programming concept, other modules have been designed. Some which appear to be helpful in use of functional modularity have been specified and simulated ([13], pp. 16-21):

- a. integer multiply and divide
- b. floating point arithmetic modules
- c. trigonometric modules: Sine, Arcsine, Arctangent

10. RTMs have been used in simulations to directly simulate FORTRAN functions such as assignment, branching, and DO loops. [13], p. 21.

11. Triangulation

RTM programs have been written for the triangulation problem; "computation of the lead angle of intercept for a defensive weapons system (airborne or surface) on an approaching target." [13], p. 22. The algorithm has two steps:

1. Given two marks (target sightings with range, bearing and time), determine target's relative course and speed.

2. Given relative course and speed of target and relative speed of weapon, determine intercept bearing.

4.3 Functional Program Modules

The choice of which functions to implement as Register Transfer Modules was partly determined by the state of MSI technology, and partly by the intended marketing audience. When attempting to generalize the best features of these concepts and imagine improvements, it is sometimes difficult to maintain proper use of the RTM terms as copyrighted. Furthermore, when attempting to unify the fundamentally important concepts which are common to RTMs and many other systems, the specific nature of RTMs, especially the actual PDP-16 modules, are less than vital to the discussion.

We therefore adopt the phrase, Functional Program Modules (FPMs) to stand for a set of functional building blocks applicable for direct implementation of programs.

Approximately 20 different modules were initially postulated in the RTM system [1], and the list in a preceding section of simulated modules includes several additional ones. It is to be expected that early phases of development of any such system will initially lead to a large number of different modules (50 might be a reasonable guess), in order to satisfy a wide range of applications. As experience grows, however, two trends will reduce this number significantly. One is usage. The modules which are poorly conceived will find less and less use, and will disappear in favor of the better ones. The other experience trend will result from the maturation of the design level and adaptation of the FPM approach to better fit this level. Thus as the most

suitable defining terms, the most expressive language and the best organizational structures appear, a smaller, more effective set of modules will develop simultaneously.

It must be recognized that the modules have been chosen in order to give a sufficient and expressive set in which to express designs. No other effort, such as a search for a minimal set, has been allowed to directly influence the choice of modules; this certainly would detract from the attempt at descriptive richness in the language.

A later development can be expected which might reduce the number. A reasonable expectation might be four to ten modules for a certain class of designs; a higher level of description might use a still smaller number of complexes of these modules as basic units, but would incur some inefficiency in the form of unused, redundant capabilities.

For the present, it would seem that the modules to be developed, designed or simulated should include ones to perform essentially all of the operations under consideration as machine instructions for future digital computers, ([11] for example) such as:

- complete serial and parallel control instructions

- subroutining and stacking

- synchronization functions

- conversion (fixed/floating, scaling, rounding; polar/cartesian coordinates; time scales; BCD/binary and various graphic and communication codes)

- vector, matrix and higher-order array operations

- circular and hyperbolic functions and their inverses

- structured data operators

- logarithms and power functions

- common algebraic expression forms such as $ax+b$, polynomials, $\sum x^2$ etc.

As experience grows in any given application, common substructures will emerge; these often-used substructures will perhaps become standard modules themselves. Also in the course of developing the best set of functional program modules, certain simplifications will occur. For example, most of the conversions listed above can be efficiently organized as a table lookup procedures, with the only difference being the contents of a read-only memory.

It will be worthwhile, part way through this development, to devote some effort to developing a "minimal" set of modules. It may well be, for example, that the same single module with different ROMs will perform each of the required functions with acceptable efficiency - this could be one version of a "universal functional module."

5. Applications and Conclusions

5.1 Applications of Functional Program Modules

As described in this report, Functional Program Modules have already proved to be effective in construction of a small number of data systems with intermediate complexity. The principles employed appear promising for further and wider applications in several ways.

Perhaps the most important feature of use of the modules is the manner in which they interact with the rest of the design process. They promise to be very effective as components in a unified design system which is highly automated and thus extremely fast in responses to new design requirements or reconfigurations. The actual implementation of FPMs as a set of production units is somewhat less important here than the concept of a set of functions which can be used as elementary design components at the register-transfer level and above. It may well be that the best use of physical FPMs is as a set of breadboarding modules for advanced development; a more compact realization might be developed for high-volume production.

The FPMs not only interface well with the production-oriented part of digital systems design, but also lend themselves to cooperation with the analytical stages. The modules so far conceived are, as demonstrated in Section 4.2.3, very well suited for use in simulations.

Experimental module designs and system configurations can easily be simulated for the purpose of evaluating a design and its analysis. For example, a simulated system might be configured and run with realistic inputs in order to discover or verify important characteristics about job-related statistics and parameters.

Interfaces and special-purpose units will be easily configured with FPMs. The FPM design approach will therefore be useful not only for configuration of overall system structures but also for validation of design and fabrication models of subsystems (which may or may not be designed according to FPM techniques). Simulated, breadboarded or manufactured subsystems can be connected to, and operated with, FPM-designed test beds for development validation.

In addition to their use as functional data processing elements, FPMs appear to be well suited for use as the control elements for cooperating sub-processors. Despite the compelling reasons for an integrated design approach to electronics systems, particularly for small ships, it appears that many of the near-future efforts will involve separate development of sensor, communication, computation and control functions. The best that can be done in such a situation is to integrate the many separate systems by a common control network in such a way as to promote their cooperation. FPMs are well suited for such a coordinating role, where some processes operate in parallel, others in serial fashion, some explicitly synchronized and others merely communicating. As described in the preceding sections and in the references, the very nature of the program design process has resulted in an effect control structure for FPMs.

A portion of the FPM development effort should be devoted to analysis and testing of means for using FPM networks as programming structures for separately developed subsystems.

5.2 Conclusions

The use of Functional Program Modules as a set of components for digital systems design has been shown to be an effective and versatile concept, whether the modules are used as a convenient conceptual unit for design or as an actual physical unit for the design-to-construction continuum, or as a control programming structure.

The feasibility of their use with regard to modern packaging techniques and emerging component technologies has not been fully explored. FPM development has not yet been oriented toward producing example applications which directly compete either against high-volume production of gate-level designs or against use of general-purpose computers of any size. Further work along these lines is needed.

The gradual evolution toward effective use of modular, functional programming will continue, with or without any special attention from the Navy directed toward the modules themselves. But the evolution will be gradual, and it will proceed through a long period in which designers in each particular phase of signal and information processing and in each industrial organization will develop their own peculiar approach. From observations of recent activities concerned with general purpose computers used in tactical data systems, it seems there is a substantial danger that some one or several of these approaches might be frozen as ad hoc, through official, Navy standards. Or worse yet, there may be many: a weapons system standard, a radar standard, or sonar standard, an optics standard, a communications standard, ... etc. Each of these areas has its own particular concerns, and would naturally evolve a somewhat different organization of modules. The only hope of compatibility in function would be through externally enforced interface specifications. Clearly this is less than the best of situations.

Much better is the prospect that is presented by the case for a general development plan for a set of Function Program Modules. The FPMs developed in this manner would be effective for a wide range of naval electronics systems. These modules will necessarily be individually efficient for use in each of the several operational specialties, but unified development from a common base will naturally and easily lead to cooperating interfaces and minimal diversity in logistic and maintenance support.

Most of this report has considered FPMs as modules which perform as well as control their functions. There are corresponding data operators and program control units. The latter are the most important, and may have some applications to the control of independently developed signal or data system processors. The most important feature of the FPM method is that it presents a unified approach to the problem of creating designs which include cooperating components within a complex system.

References

1. C. G. Bell, J. L. Eggert, J. Grason, P. R. Williams, "The description and use of Register - Transfer Modules (RTMs)," IEEE Transactions on Electronic Computers, Short Notes, C - 21, 5 May 1972, 495-500.
2. C. G. Bell, J. Grason et al. "The design, description and use of DEC Register Transfer Modules (RTM)," Dept. of Computer Science, Carnegie - Mellon University, 14 October, 1970.
3. C. G. Bell and A. Newell, "The PMS and ISP descriptive systems for computer structures," Proc. AFIPS 1970 SJCC, 33, 351-374.
4. C. G. Bell, A. Newell, Computer structures: readings and examples, McGraw-Hill, 1971.
5. M. A. Breuer, "Recent developments in design automation," Computer, 5, 3, May/June, 23-25.
6. Y. Chu, Introduction to Computer Organization, Prentice-Hall, 1970.
7. Y. Chu, O. R. Pardo, J. Yeh, "A methodology for unified hardware-software design," Tech. Rep. 70-107, University of Maryland Computer Science Center, College Park, Maryland, Jan. 1970.
8. A. B. Marcovitz, E. J. Schweppe, An Introduction to Numerical Methods Using the MAD Language, The Macmillan Company, New York, 1966.
9. D. L. Parnas, "A technique for software module specification with examples," Carnegie-Mellon University Computer Science Department technical report, March, 1971.
10. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Carnegie-Mellon University Computer Science Department technical report, August, 1971.
11. Draft Copy of, Raytheon Missile Systems Division, A Deerfield et al., "Interim report for arithmetic and control logic design study," report BR-7034, May 1972.
12. R. L. Russo, "Design automation", Computer, 5, 3, May 1972, p. 18.
13. A. H. Spruell, Jr., "The construction of digital computers using register transfer modules," U. S. Naval Postgraduate School Master's thesis, Sept. 1971.
14. PDP-16 Computer designers handbook, Digital Equipment Corporation, 1971.
15. S. L. Margolis, "Computer program fault detection and correction," U. S. Naval Postgraduate School Master's thesis, June 1972.

Distribution List

Defense Documentation Center Cameron Station Alexandria, Virginia	12
Library Naval Postgraduate School Monterey, California 93940	2
Naval Electronics Laboratory Center Research Library, Code 6700 San Diego, California 92152	2
NELC Mr. W. J. Dejka, Code 4300 San Diego, CA 92152	8
NELC Mr. S. Snyder, Code 4000 San Diego, CA 92152	1
NELC Mr. H. T. Mortimer, Code 0220 San Diego, CA 92152	1
NELC Mr. M. Lamendola, Code 5200 San Diego, CA 92152	1
NELC Mr. W. Loper, Code 5200 San Diego, CA 92152	1
NELC Dr. R. Kolb, Code 3300 San Diego, CA 92152	1
NELC Mr. E. E. McCown, Code 3100 San Diego, CA 92152	1
NELC Mr. H. F. Wong, Code 3200 San Diego, CA 92152	1
NELC Dr. P. C. Fletcher, Code 2000 San Diego, CA 92152	1

Distribution List (cont.)

Dean of Research Administration Naval Postgraduate School Monterey, California 93940	2
Professor Sydney R. Parker Chairman, Dept. of Electrical Engg. Naval Postgraduate School Monterey, CA 92940	2
Professor V. Michael Powers Dept. of Electrical Engineering Naval Postgraduate School Monterey, CA 93940	4
Professor Gordon H. Syms Dept. of Mathematics Naval Postgraduate School Monterey, CA 93940	1

DOCUMENT CONTROL DATA - R & D

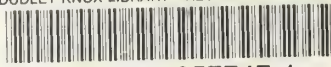
Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified.

1. ORIGINATING ACTIVITY (Corporate author) Naval Postgraduate School Monterey, California 93940		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP	
3. REPORT TITLE Functional Program Modules (FPMs) and Digital Systems Design			
4. DESCRIPTIVE NOTES (Type of report and, inclusive dates) Technical Report 1971-72			
5. AUTHOR(S) (First name, middle initial, last name) V. Michael Powers			
6. REPORT DATE 20 July 1972		7a. TOTAL NO. OF PAGES 53	7b. NO. OF REFS 15
8a. CONTRACT OR GRANT NO. Work Request No. 2-9104, amendment		9a. ORIGINATOR'S REPORT NUMBER(S) NPS-52PW72071A	
8b. PROJECT NO. number one, April 17, 1972.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY	
13. ABSTRACT Design of digital electronic systems for a range of applications such as the functions of small ships requires a unified approach which does not make a priori choices between hardware and software. Throughout the several levels of digital system structure, the design procedure is portrayed as a task of preparing a program and translating the program into successively different languages. One language is the language of Functional Program Modules. These modules are versatile, effective, program control and operative modules which can easily be configured (programmed) and can be used with methods which automatically produce assembly and maintenance information. Examples, extensions and applications of FPMs are discussed.			

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
functional program modules register transfer modules digital system design hardware design software design automatic design						

U146924

DUDLEY KNOX LIBRARY - RESEARCH REPORTS



5 6853 01057747 1

014092